Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 2: this Friday @ 11:59 pm
  - Lab 1: next Tuesday @ 11:59 pm
  - Assignment 1: Friday Feb 17th @ 11:59 pm
- You can view correct answers for homework questions
- Jupyterhub server for testing out small snippets of code
  - jupyterhub.sci.pitt.edu
  - Use your Pitt username and password
  - Has to be either on campus or over Pitt VPN
    - You can connect to Pitt VPN using the PulseSecure program; insturctions available on Pitt IT website
- TAs student support hours available on the syllabus page

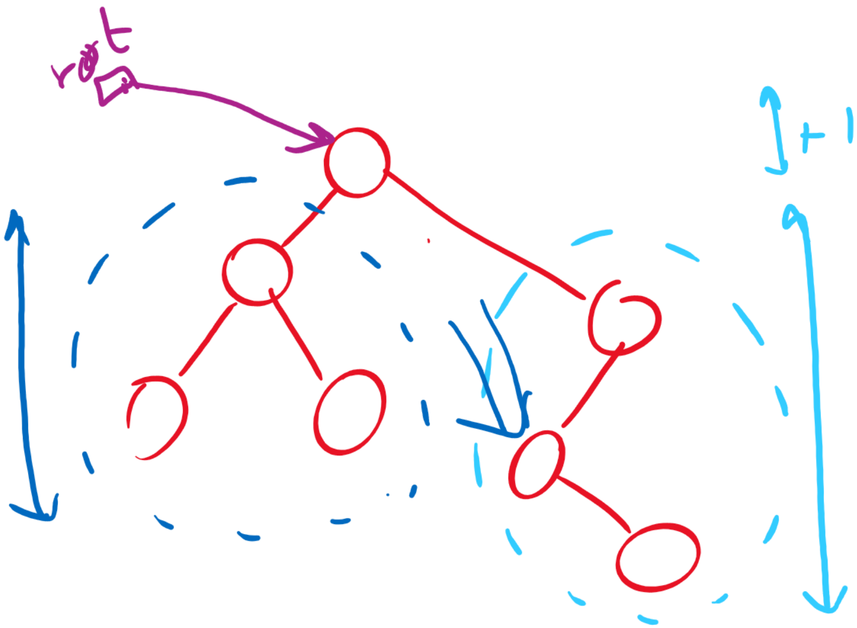
Previous lecture

- ADT Tree
  - Binary Tree
- BinaryTree implementation
  - BinaryNode

This Lecture

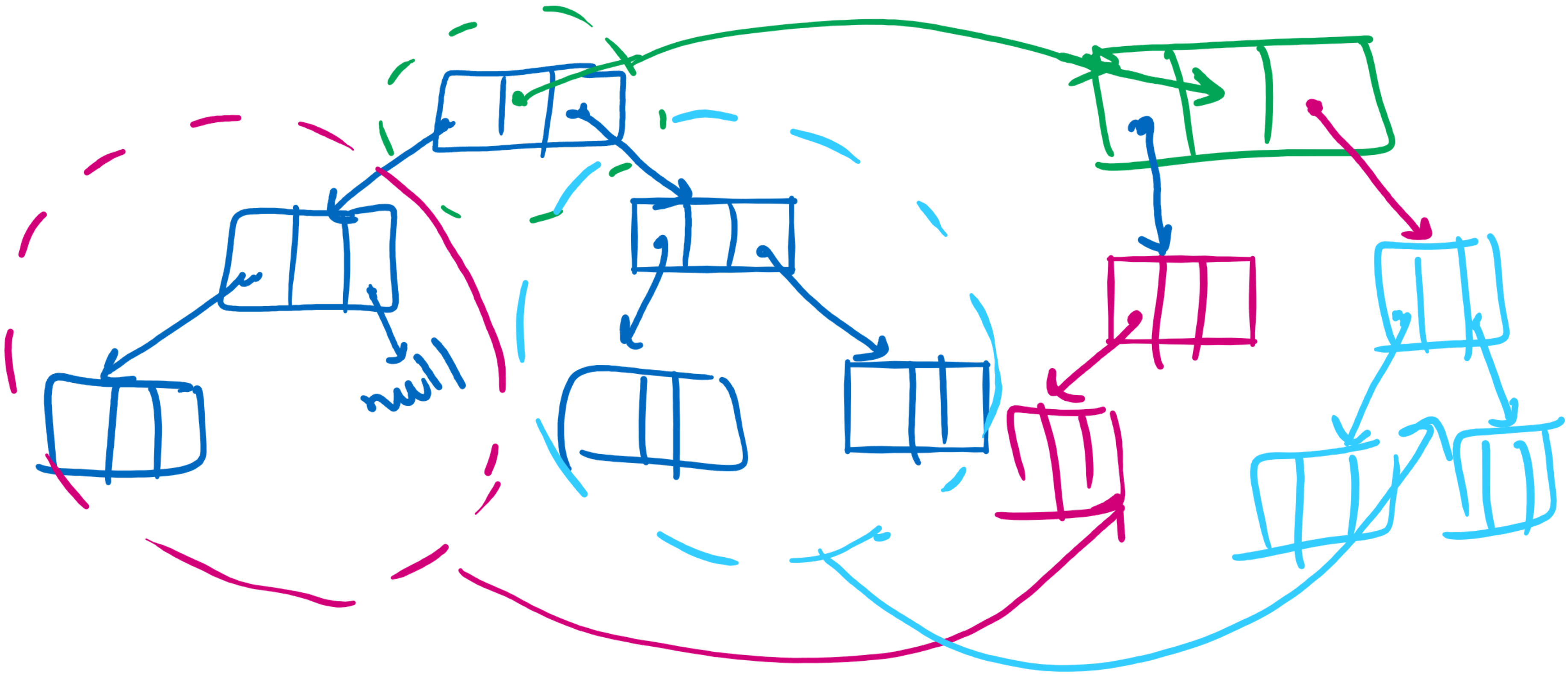
- Binary Tree Implementation
  - buildTree
  - tree traversals
- Binary Search Tree
  - How to add and delete
- Runtime of BST operations
  - Find, add, delete

Another implementation of getHeight




```
int getHeight(BinaryNode<T> root){
    int lHeight=0;
    int rHeight=0;
    if(root.left != null)
        lHeight=getHeight(root.left);
    if (root.right != null)
        rHeight= getHeight(root.right);
    return Math.max(lHeight, rHeight)+1;
}
```

BinaryNode.copy



buildTree method

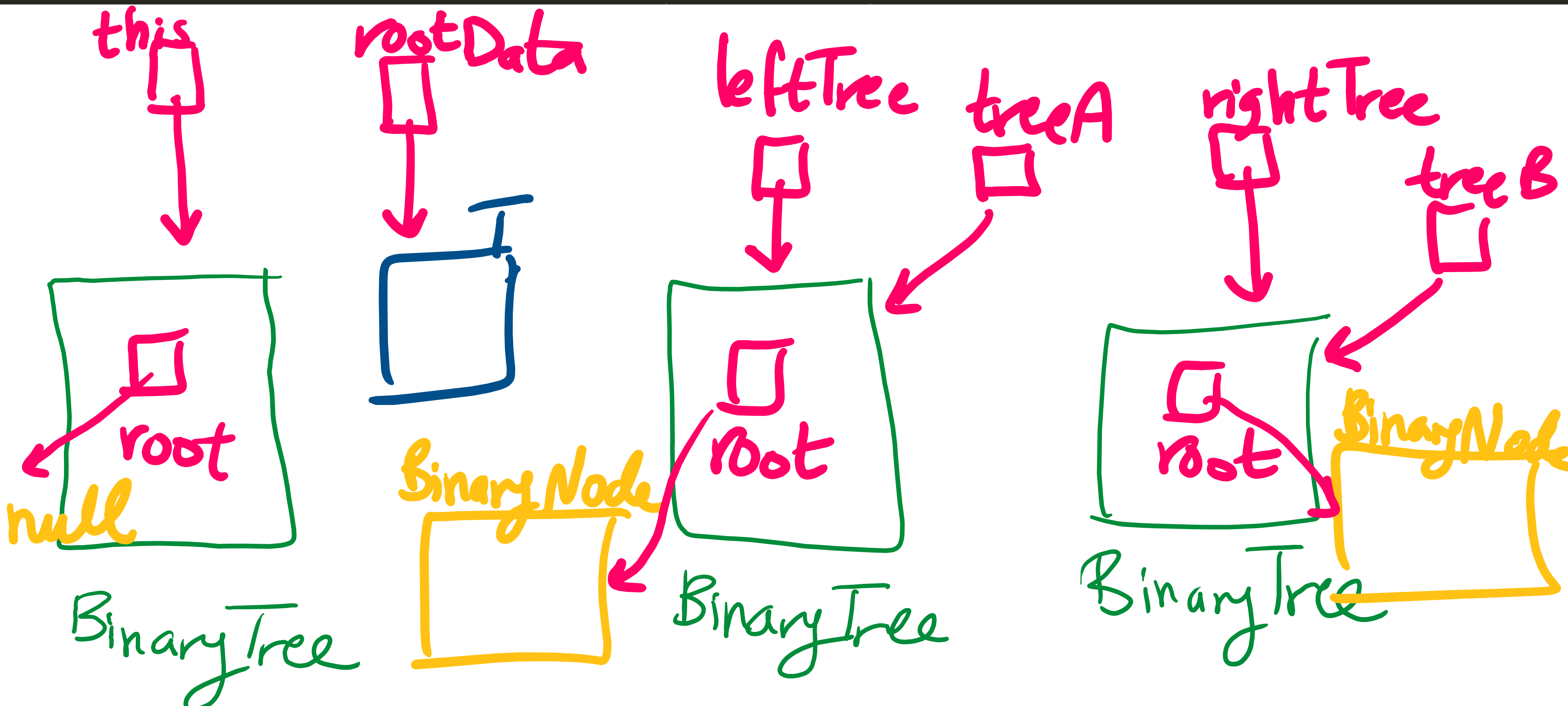
```
private void privateBuildTree(T rootData, BinaryTree<T> leftTree,
                              BinaryTree<T> rightTree){
```

Let's draw a picture of the before state

- Given the call

```
privateBuildTree(data, treeA, treeB);
```

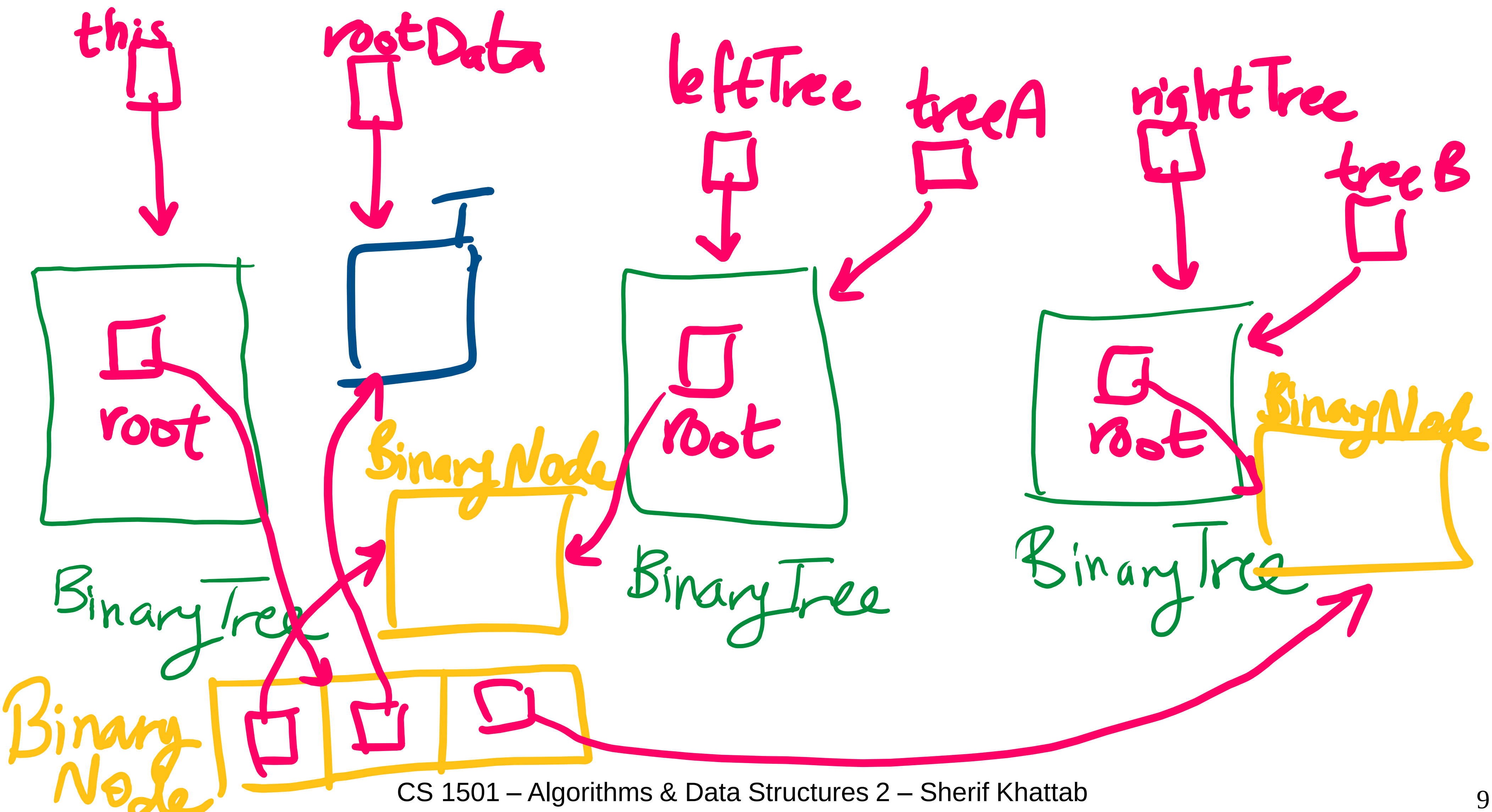
```
private void privateBuildTree(T rootData, BinaryTree<T> leftTree,
                              BinaryTree<T> rightTree){
```



Let's draw a picture of the after state

```
privateBuildTree(data, treeA, treeB);
```

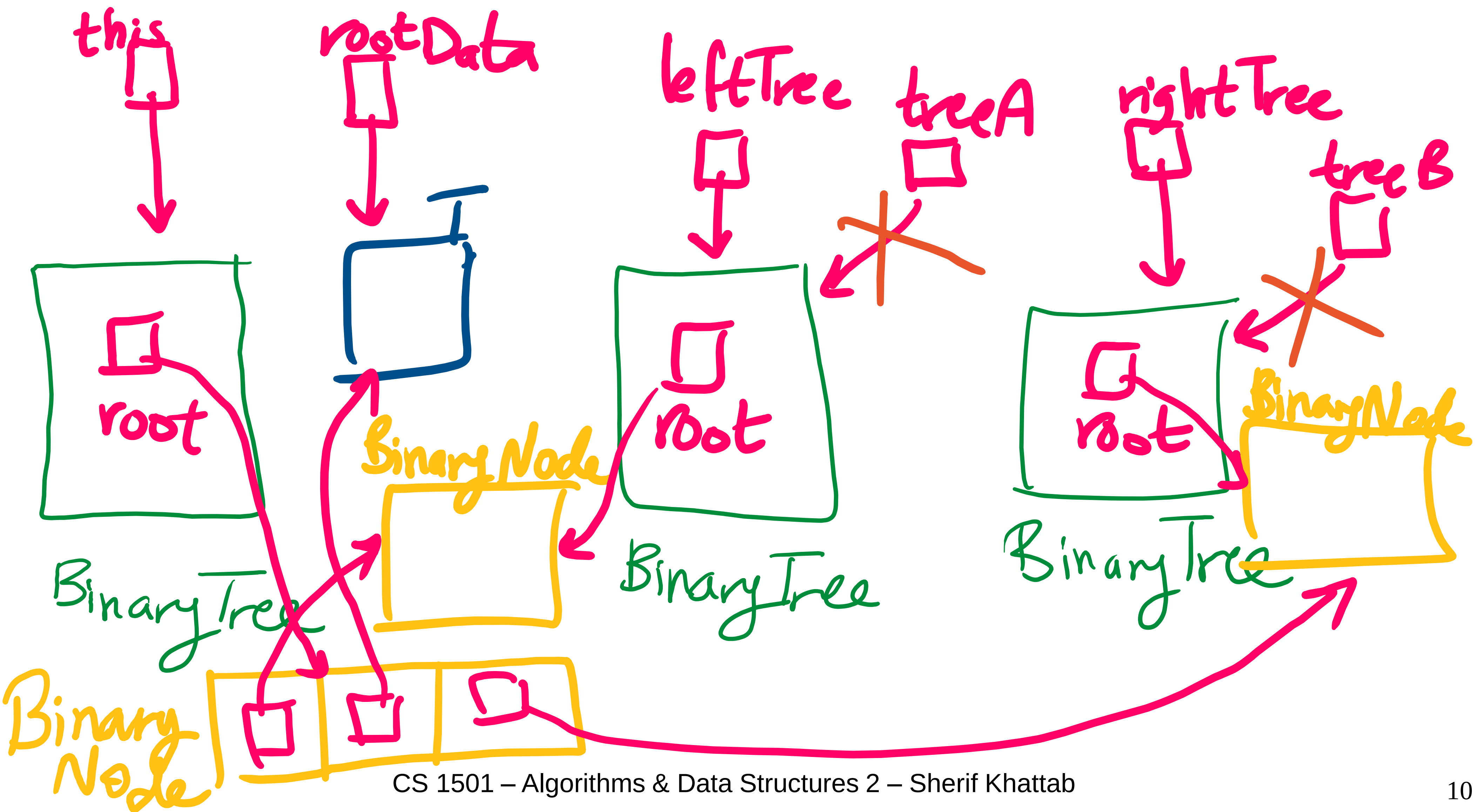
```
private void privateBuildTree(T rootData, BinaryTree<T> leftTree,
                              BinaryTree<T> rightTree){
```



Let’s draw a picture of the after state

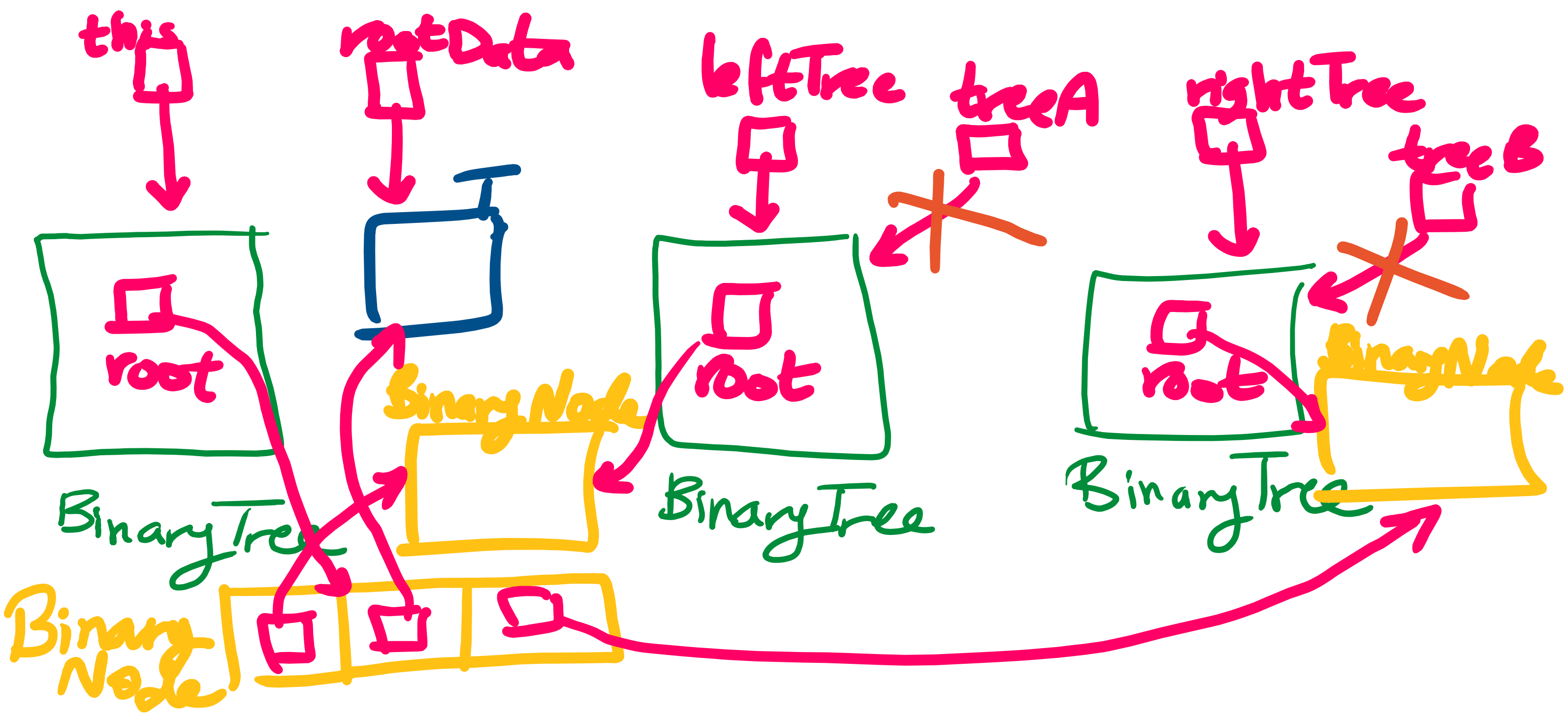
Need to also Prevent client direct access to this

`treeA` shouldn’t have access this.root.left (same for treeB)



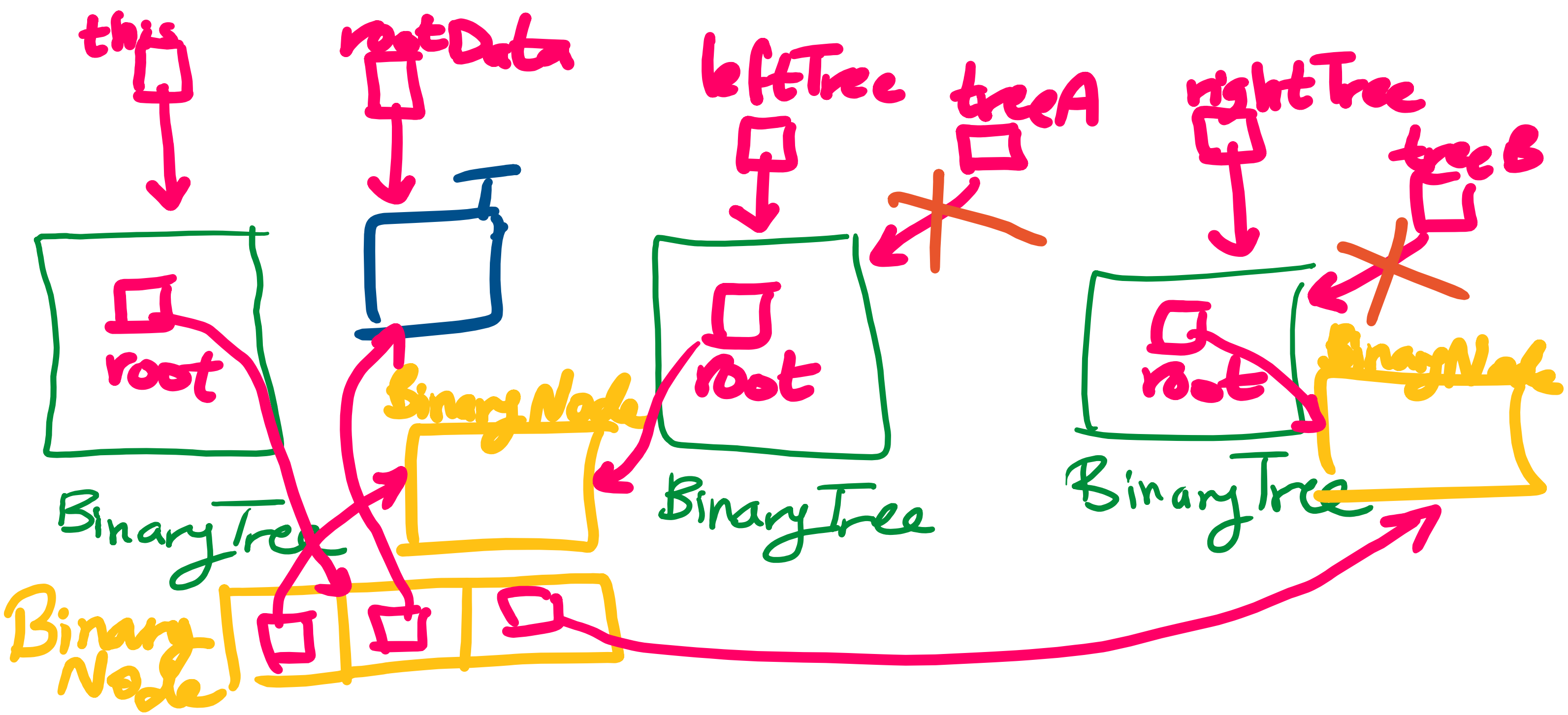
Main logic

- `root = new BinaryNode<>(rootData);`
- `root.left = leftTree.root;`
- `root.right = rightTree.root;`
- How to prevent client access?



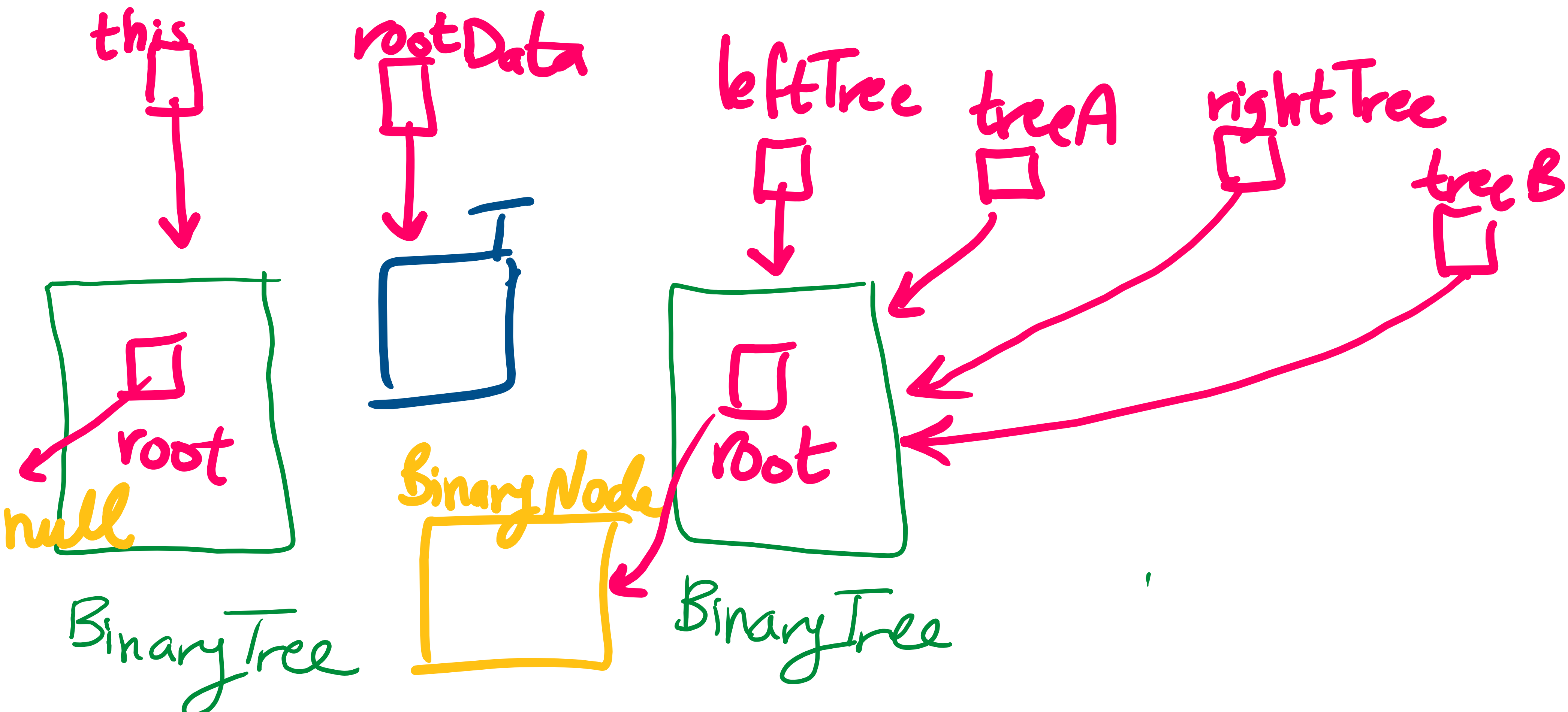
How to prevent client access?

- treeA = treeB = null; //is that possible?
- leftTree = rightTree = null; //would that work?
- leftTree.root = null; rightTree.root = null;



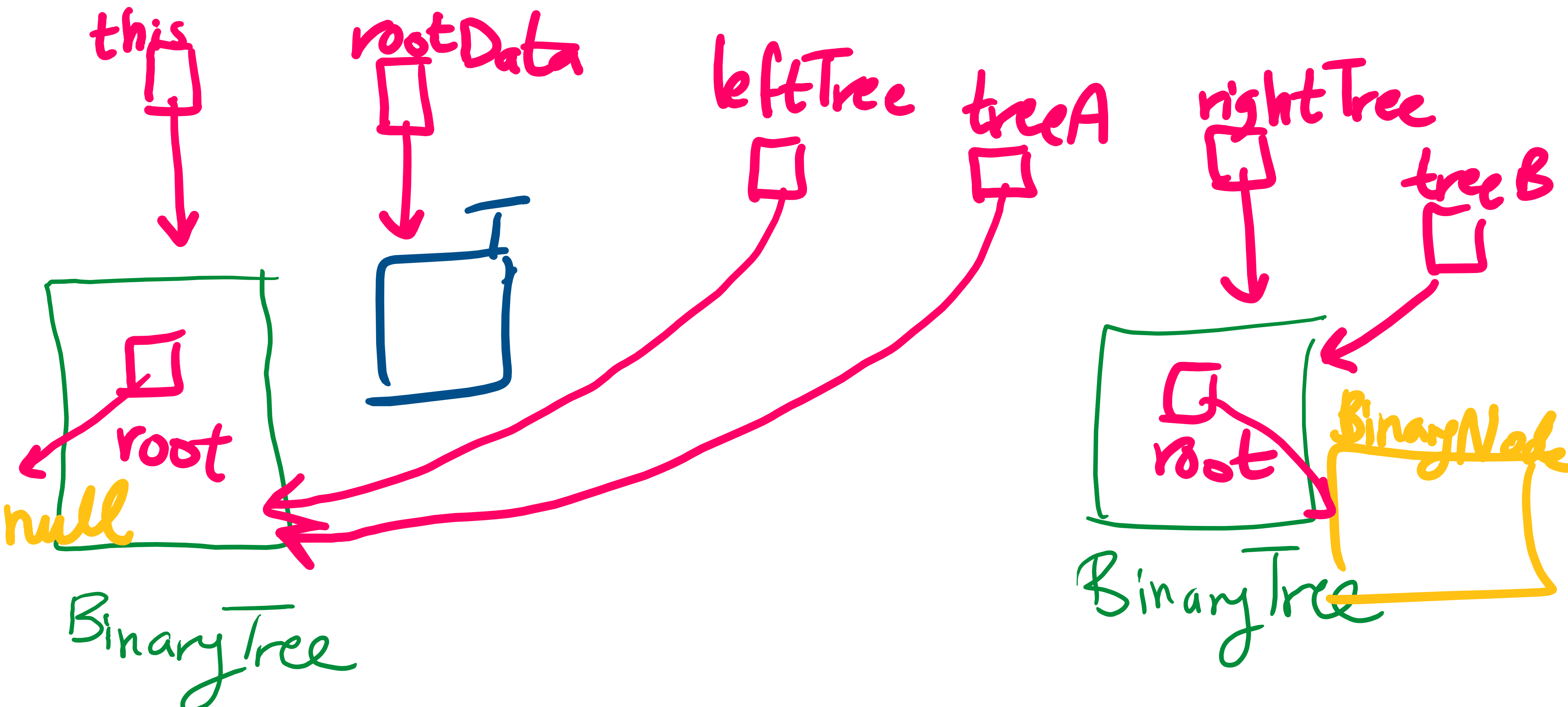
Special case: treeA == treeB

Need to make a copy of leftTree.root



Special case: treeA == this or treeB == this

Need to be careful before leftTree.root = null and rightTree.root = null



Tree Traversal Methods

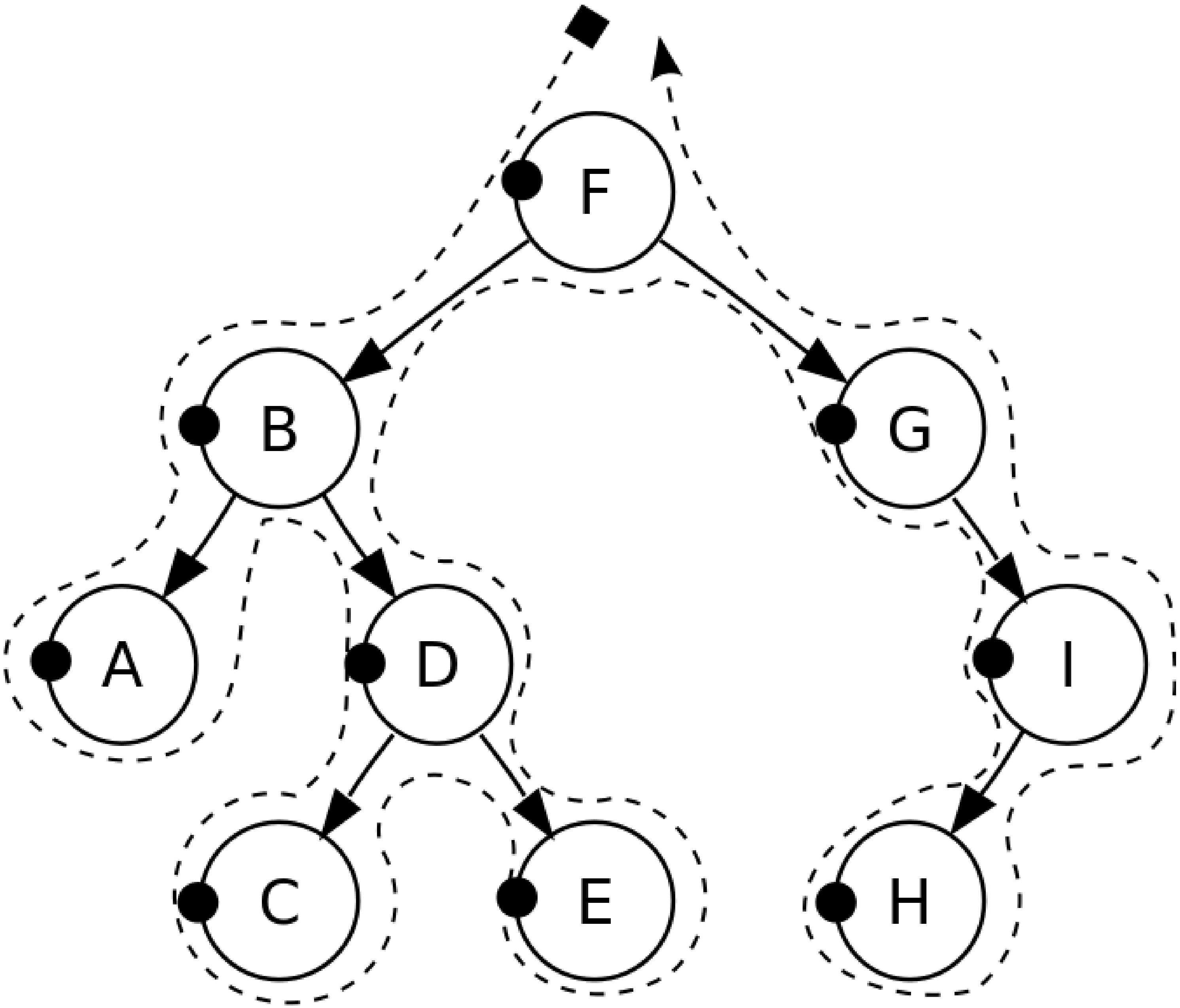
- How to traverse a Binary Tree
  - General Binary Tree
    - Pre-order, in-order, post-order, level-order

Traversals of a General Binary Tree

- Preorder traversal
  - Visit root **before** we visit root's subtree(s)

Pre-order traversal

- F
- B
- A
- D
- C
- E
- G
- I
- H



Pre-order traversal implementation

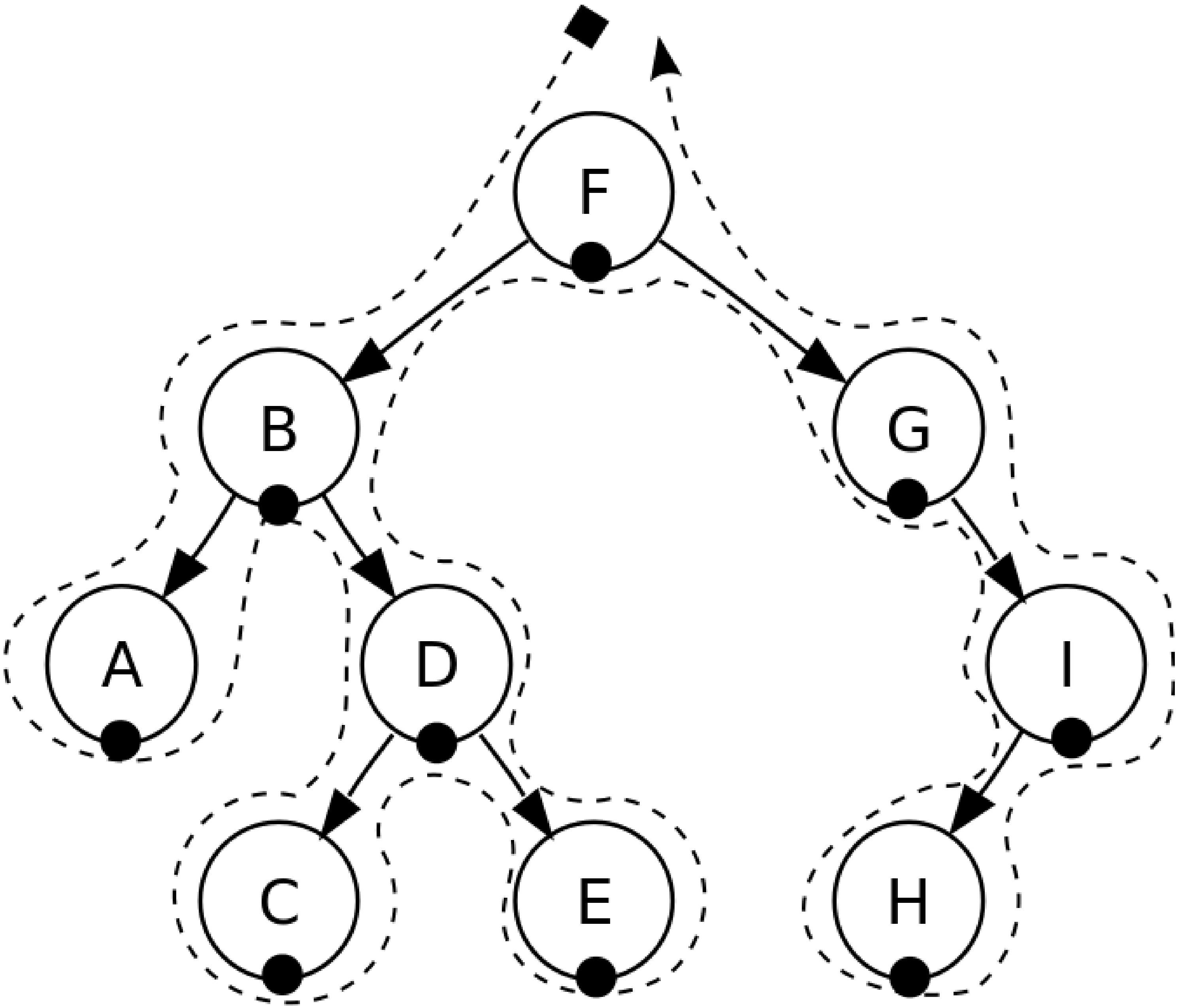
```
void traverse(BinaryNode<T> root){
    if(root != null){
        System.out.println(root.data);
        traverse(root.left);
        traverse(root.right);
    }
}
```

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before we visit root's subtrees
- In-order traversal
  - Visit root of a binary tree **between** visiting nodes in root's subtrees.
  - left then root then right

In-order traversal

A
B
C
D
E
F
G
H
I



In-order traversal implementation

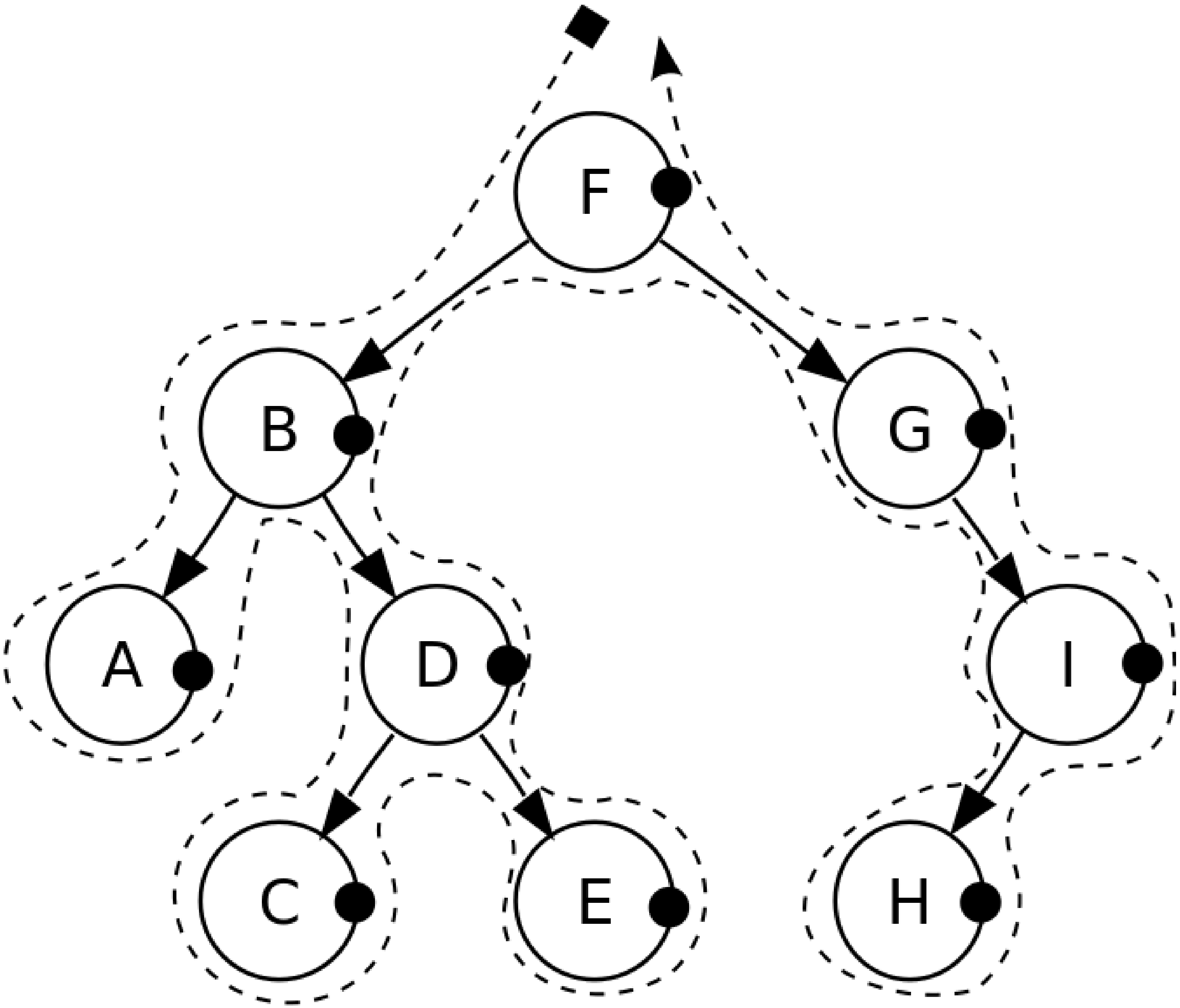
```
void traverse(BinaryNode<T> root){
   if(root != null){
      traverse(root.left);
      System.out.println(root.data);
      traverse(root.right);
    }
}
```

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before we visit root's subtrees
- Inorder traversal
  - Visit root of a binary tree between visiting nodes in root's subtrees.
- Postorder traversal
  - Visit root of a binary tree after visiting nodes in root's subtrees

Post-order traversal

A
C
E
D
B
H
I
G
F



Post-order traversal implementation

```
void traverse(BinaryNode<T> root){
    if(root != null){
        traverse(root.left);
        traverse(root.right);
        System.out.println(root.data);
    }
}
```

Traversals of a Binary Tree

- Preorder traversal
  - Visit root before we visit root’s subtrees
- Inorder traversal
  - Visit root of a binary tree between visiting nodes in root’s subtrees.
- Postorder traversal
  - Visit root of a binary tree after visiting nodes in root’s subtrees
- Level-order traversal
  - Begin at root and visit nodes one level at a time
  - We will see the implementation when we learn Breadth-First Search of Graphs

Tree Search Take 1

- *Traverse* every node of the tree
  - Is the key inside the node equal to the target *key*?
- How can we traverse the tree?

Tree Search Take 1

What is the runtime?

Can we do better?

Can we traverse the tree more intelligently?

Tree Search Take 2: Binary Search Tree

- Search Tree Property
  - left.data < root.data < right.data
  - Holds for each subtree
  - In Java:
    - root.data.compareTo(left.data) > 0 &&
    - root.data.compareTo(right.data) < 0

Binary Search Tree

- Search Tree Property
  - For each node in the tree:
    - The data of the node is larger than the data in all nodes in the node’s left subtree and
    - The data of the node is smaller than the data in all nodes in the node’s right subtree

Let’s build a Binary Search Tree

- Work in groups of 2-3 students
- Add the following integers to a Binary Search Tree in **<u>the following order</u>**:

10, 8, 17, 7, 5, 20, 15, 16, 4

Reflect on the steps that you followed

- How did you add 4 to the tree?
- What steps did you follow?

10, 8, 17, 7, 5, 20, 15, 16, <u>4</u>

BST: How to add?

- How to add a data item *entry* into a BST rooted at *root?*
- What if root.data.compareTo(entry) == 0?
- What if root.data.compareTo(entry) < 0?
  - Move left or right?
  - What if no child?
  - What if there is a child?
- What if root.data.compareTo(entry) > 0?
  - Move left or right?
  - What if no child?
  - What if there is a child?
- What if I tell you that you have a friend who can add into a BST.
  - How can you use the help of that friend?

Let’s see the code for adding into a BST

- Available online at:
  - https://cs1501-2231.github.io/slides-handouts/CodeHandouts/TreeADT/Slides
  - The slides are under the CodeHandouts/TreeADT/slides folder in the handout repository
  - https://github.com/cs1501-2231/slides-handouts

Let’s build a Binary Search Tree

- Work in groups of 2-3 students
- Add the following integers to a Binary Search Tree in **<u>the following order</u>**:

4, 5, 7, 8, 10, 8, 15, 16, 17, 20

Reflect on the steps that you followed

- How many comparisons did you have to make to add 20?

4, 5, 7, 8, 10, 8, 15, 16, 17, <u>20</u>

Run-time of BST operations

- For add, # comparisons = d, where *d* is the depth of the new node
- For search miss, # comparisons = d, where *d* is the depth of the node if it were in the tree
- For search hit, # comparisons = 1+d, where *d* is the depth of the found node
- On average, node depth in a BST is O(log n)
  - *n* is the number of data items
  - Proof in Proposition C in Section 3.2 of Sedgewick Textbook
- In the worst case, node depth in a BST is O(n)

Let's switch to delete!

- Work in groups of 2-3 students
- In the Binary Search Tree that you built out of **the following order**:

10, 8, 17, 7, 5, 20, 15, 16, 4

- How would you delete 4?
- How would you delete 5?
- How would you delete 10?

BST: How to delete?

- Three cases
  - Case 1: node to be deleted is a leaf
    - Easiest case!
    - Pass back null to the node's parent
  - Case 2: node to be deleted has one child
    - Pass back the child to the node's parent to adopt instead of the node

BST: How to delete?

- Three cases
  - Case 3: node to be deleted has two children
    - Difficult to delete the node!
      - It has two children, so parent cannot adopt both
    - Let’s try to replace the data in the node
      - We still want to maintain the search tree property
      - What are our options?
    - Replace node’s data by its successor
      - largest item in left subtree
    - or by its predecessor
      - smallest item in right subtree
    - Delete the node that we selected in the previous step
      - Has to have at most one child
      - Why?

Let’s see the code for deleting from a BST

- Available online at:
  - https://cs1501-2231.github.io/slides-handouts/CodeHandouts/TreeADT/Slides
  - The slides are under the CodeHandouts/TreeADT/slides folder in the handout repository
  - https://github.com/cs1501-2231/slides-handouts